13 | 分布式锁Chubby（二）：众口铄金的真相

2021-10-20 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 24:08 大小 22.11M

你好，我是徐文浩。

上一讲里，我为你解析了两阶段提交和三阶段提交是怎么回事儿。相信你和我一样，对这两种解决方案都不太满意。虽然它们可以帮助我们实现一个分布式的事务，但同时也有着很明显的缺陷：这两个都是一个“单点”特别明显的系统，一旦作为单点的“协调者”出现网络问题或者硬件故障，整个系统就没法运行了。另一方面，如果你为了一致性，选择了两阶段提交，那么整个系统就是“同步阻塞”的，意味着整个系统的性能也会受到很大的影响。

单点故障和同步阻塞，使得所谓的“分布式”系统的优势完全没有发挥出来。所以，我们需要一个对单点没有依赖，并且不是同步阻塞的解决方案。而这个方案呢，就是我今天要给你解读的 **Paxos 算法**。并且在这节课里，我还会带你深入来看一下我们一般在数据库领

域所说的“事务性”到底是怎么回事儿、所谓的 ACID 又是什么意思。特别是当我们处于一个分布式的环境下，又会面临什么样的挑战。

在学完这节课之后，我相信你会和我一样，搞清楚这三件事情：

数据库领域的 ACID 具体是指什么，里面的隔离性具体分成哪几种隔离等级？

分布式系统场景下，事务怎么来保障系统的“隔离性”？

Paxos 算法，究竟是怎么一回事儿？

而在搞清楚这三件事情之后，你也就掌握了足够的分布式事务和分布式一致性的基础知识，这也就意味着，你能够读懂 Chubby 的论文了。

理解 ACID

在上一讲里，我们提到的两阶段提交，是想要解决 Master 和 Backup Master 完全同步的问题。事实上，这个方式适用于所有的“分布式事务”问题。我们不是只能让两个参与者写入完全相同的数据，两阶段提交也完全可以在两个不同的物理数据库里，插入不同的数据。

比如，我们上一讲里的买卖房屋的例子，买房的银行账户的金额变动和卖房的银行账户的金额变动，可能就在两个不同的数据库。但是在实际银行转账的时候，需要两边的银行账户同步更新，不能出现一边钱没转出去，另一边钱已经收到的情况。这个特性，也就是我们常说的数据库的**事务性**。

可想而知，事务性无论是在单机的环境下，还是分布式的环境下，都是非常重要、不可或缺的一个性质。那么，我们通常是使用 ACID 这四个字母，来描述事务性需要满足的 4 个属性。下面我们来看看它们各自的含义。

原子性（Atomicity），也就是一个事务不能分成更小的粒度。在事务里，也就是几个操作要么同时发生，如果任何一个失败，整个就应该回滚。前面的两阶段提交，其实就是为了保障事务的原子性。但是在三阶段提交中，我们可能就会失去事务的原子性。

一致性（Consistency），这个更多是一个应用层面的属性。我们需要确保应用里面的数据是一致的。典型的例子就是我刚刚举过的，两个人买卖房屋的银行账户变动，在应用层

面，一个人减少的金额和另一个人增加的金额应该同时发生。这样数据才具有一致性，不然的话，就会凭空多出钱来或者少钱。**本质上，这个例子就是利用了数据库的事务性，实现了应用层面的数据一致性。**而数据库里的唯一约束、外键约束，都属于数据的一致性要求。

隔离性（Isolation），它需要解决的问题是，当有多个事务并发执行的时候，相互之间应该隔离，不能看到事务执行的中间状态。这个是我们今天要重点深入剖析的一个问题，也是 Paxos 算法的一个起点。

持久性（Durability），就是一旦事务成功提交，对应的数据应该保存在硬盘中，并且在硬件故障或者系统崩溃的情况下也不会丢失。持久性看起来简单，好像把数据往硬盘里一写就完事儿了，但是要真正做到并不容易。比如，如果我们的硬盘坏了，该怎么办呢？GFS 里面的 Master、Backup Master 以及数据的三份副本，都是为了保障数据的持久性。

在数据库的 ACID 属性里面，A、C 和 D 看起来是显而易见的。但是这个隔离性 I，就值得说道说道了。我们之前看到的事务的案例，是一个单独的事务。那么，**如果有两个事务想要同时发生，并且它们想要读写的数据记录之间有重合，该怎么办？**

还是回到我们之前进行房屋买卖的例子里来看，我们在一次房屋的交易过程中，有这么几个动作是在同一个事务里面发生的：

1. 检查买家的银行卡里余额是否足够；
2. 检查卖家是否实际持有房产证；
3. 把买家的付款转给卖家；
4. 把卖家的房产证作废；
5. 给买家创建一个房产证。

这样看起来很完美，既然在一个事务里面完成，满足了原子性、业务上也保障了一致性，数据库也把数据写到了硬盘上，看起来一切都很完美。

但是，如果这个卖家一房两卖怎么办呢？如果一个卖家同时和两个买家交易，会发生什么事情？在事务开始之前，房产证的确还在卖家手里，如果两个事务同时进行，在第 2 步进

行检查的时候都会通过。那么，如果事务都能够成功完成的话，就会出现一房两卖的情况，同时两个买家手里都有了房产证。

而这个问题，就是隔离性需要解决的问题。一般数据库的隔离级别，会分成四种。

未提交读（Read Uncommitted）

也就是事务之间完全不隔离。它的问题在于，在一个事务执行的过程中，读取的数据，可能会来自一个会回滚的事务。

还以卖房子为例，B 向 A 卖房子，事务执行到第 5 步，但是事务还没有提交。这个时候 C 再向 B 买同一套房子，在第 2 步检查房产证数据的时候，会觉得 A 已经有这套房子了。但是实际上，事务可能回滚，A 可能最后没有交易这个房子。基本上，未提交读不太会在真实的系统中使用，因为本质上，它会在事务中**读到脏数据**。

提交读（Read Committed）

这个隔离级别比未提交读强一些，是事务只去读取已经提交了的事务。不过在事务里，当其他事务更新了数据，事务里的多次读取可能会读到不同的结果。

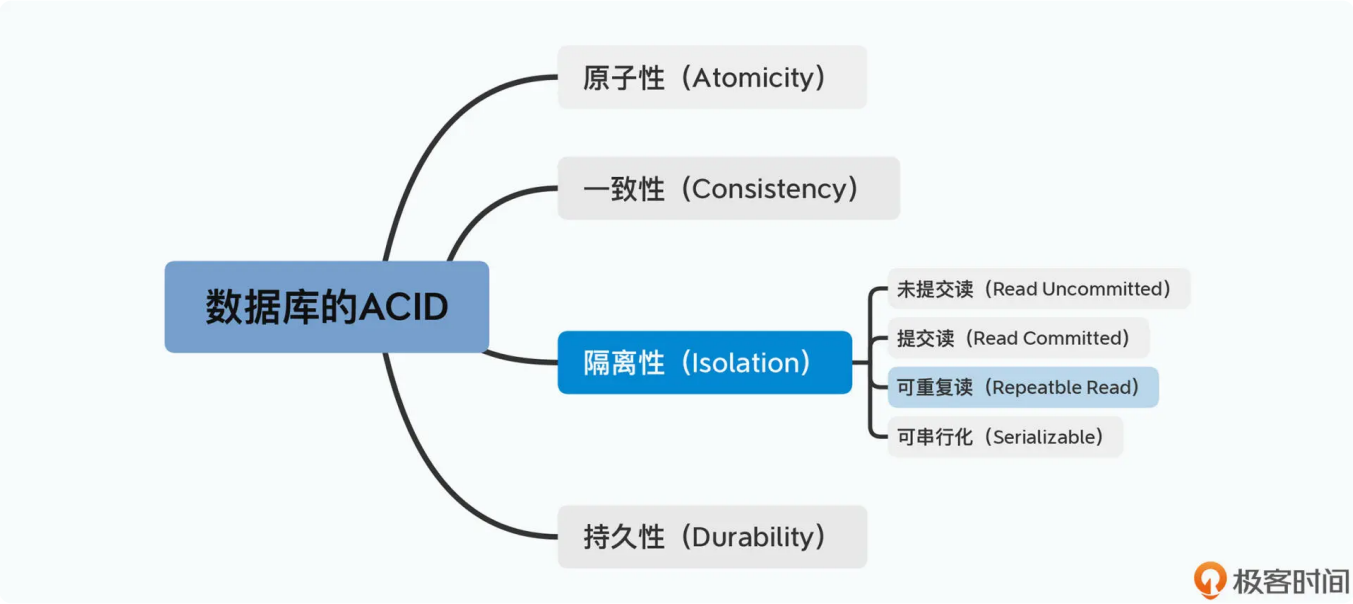
比如，一个事务里会多次读取我们的银行卡余额。如果在这个过程中，我们在其他地方刷卡消费了，那么两次读到的余额会不同。这个隔离级别一般也不太会使用，因为这样一个事务中两个完全相同的读操作，会读到完全不同的数据，也就是会出现“**幻读**”。

可重复读（Repeatble Read）

这个隔离级别是为了解决刚才的幻读问题。在可重复读的环境下，一个事务 A 一旦开始，该事务里读到的数据，都是事务开始之前已经提交的数据。如果在事务 A 的执行过程中，数据库里有新的事务 B 提交了，那么虽然数据库里的数据已经改变了，但是在事务 A 的执行过程中，还是只会当成这个事情没有发生。这个隔离方式，就会带来我们刚才所说的**一房两卖的问题**。

可串行化（Serializable）

这个是最严格的隔离方式。所有的事务，虽然提交的时候可以是并行的，但是在实际执行的过程中，在外部看来是按照一个确定的顺序一个一个执行的。就拿买卖房产的例子来说，即使卖家想要一房两卖，但是只要采用了可串行化的事务隔离方式，这两个事务在外部看起来就好像是一个一个执行的。那么，无论是哪一笔交易先执行，另一笔交易就不可能执行成功。



可以看到，很自然地，我们希望数据库的隔离级别最好都是“可串行化”的。这样，应用开发程序就会很容易写，不需要自己去考虑对特定资源加锁，只要采用事务和逻辑正确的代码，就能完成想要解决的问题。

摆脱单点故障的 Paxos

了解完了数据库的事务性，特别是里面的“隔离性”，我们发现啊，使用两阶段提交来实现 GFS 的 Master 和 Backup Master 的同步写入，好像有点杀鸡用牛刀的感觉。

为什么这么说呢？那是因为，**对于 GFS 的 Master 的操作，其实并没有有关隔离性的需求。**在这个同步复制的过程中，我们需要的并不是让 GFS 的文件系统的操作支持事务，比如“检查目录 A 是否存在，存在的话，在里面新增文件 B”。文件系统通常并不需要这三个动作保持事务性，即使需要，也是通过一个外部锁来实现。

我们对 GFS 的 Master 操作，最抽象来看，就是写日志。而 Backup Master 起到的作用，就是同步复制日志，每一条日志，都是对文件系统的一次操作。我们可以把这样的一条条操作看成是一个个事件，每一次事件，都让整个文件系统的状态进行了一次改变。所以本质上，这就是一个**状态机（State Machine）**。

而让 Backup Master 和 Master 保持同步来保持高可用性，其实就是要在两个状态机之间做同步复制。所以这个问题，也就变成了一个状态机复制问题（replicated state machine）。

在这个问题里，解决的并不是隔离性里的“可串行化”问题，而是分布式共识里的“可线性化”（Linearizability）问题。**所谓的可线性化，就是任何一个客户端写入数据成功之后，再去读取数据，一定能读取到刚才写入的数据。**

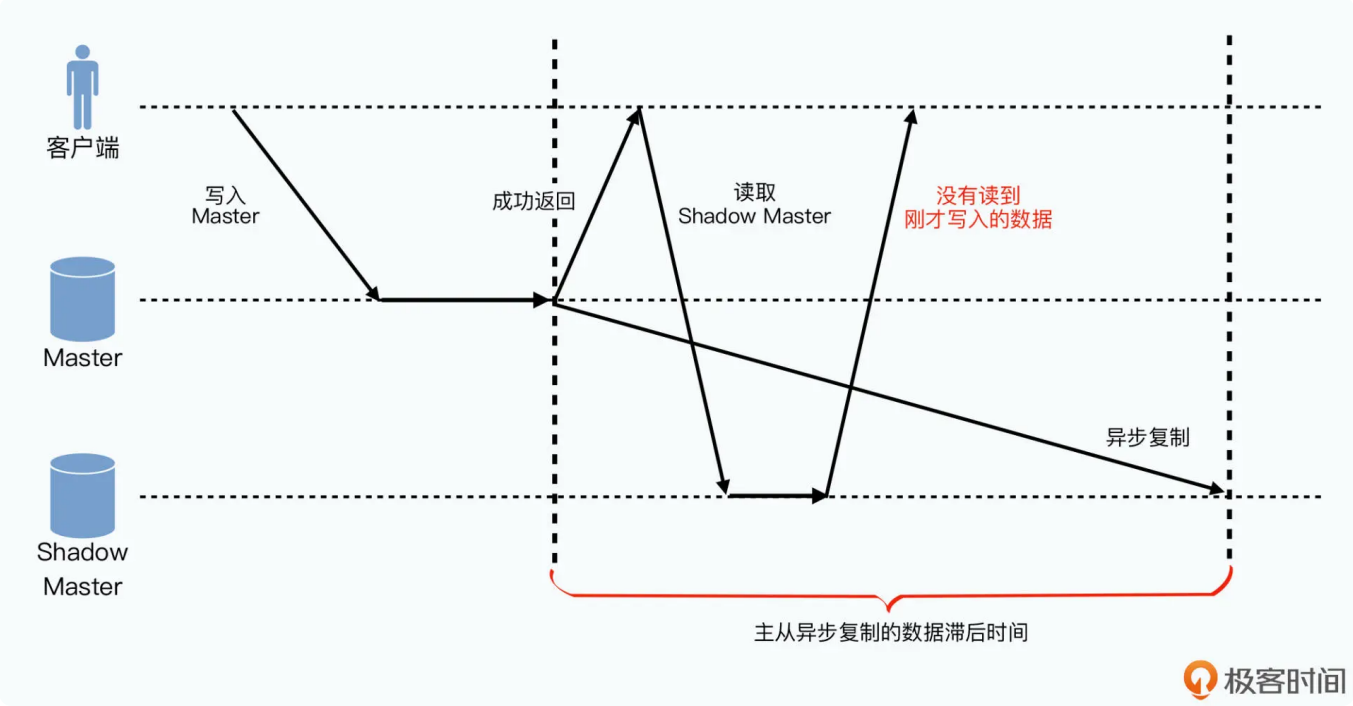
这个问题看起来好像理所当然，但是在分布式的系统中却不一定会做到。比较典型的反例，就是 GFS 中的 Shadow Master。因为 Master 到 Shadow Master 之间的数据同步是异步的，所以可能会出现这样的情况：

客户端写入数据到 Master；

Master 崩溃了；

刚才的数据还没有来得及同步到 Shadow Master；

客户端只能从 Shadow Master 读取数据，刚刚写入的数据并没有被读到。



在异步复制的情况下，我们可能会遇到线性不一致的问题

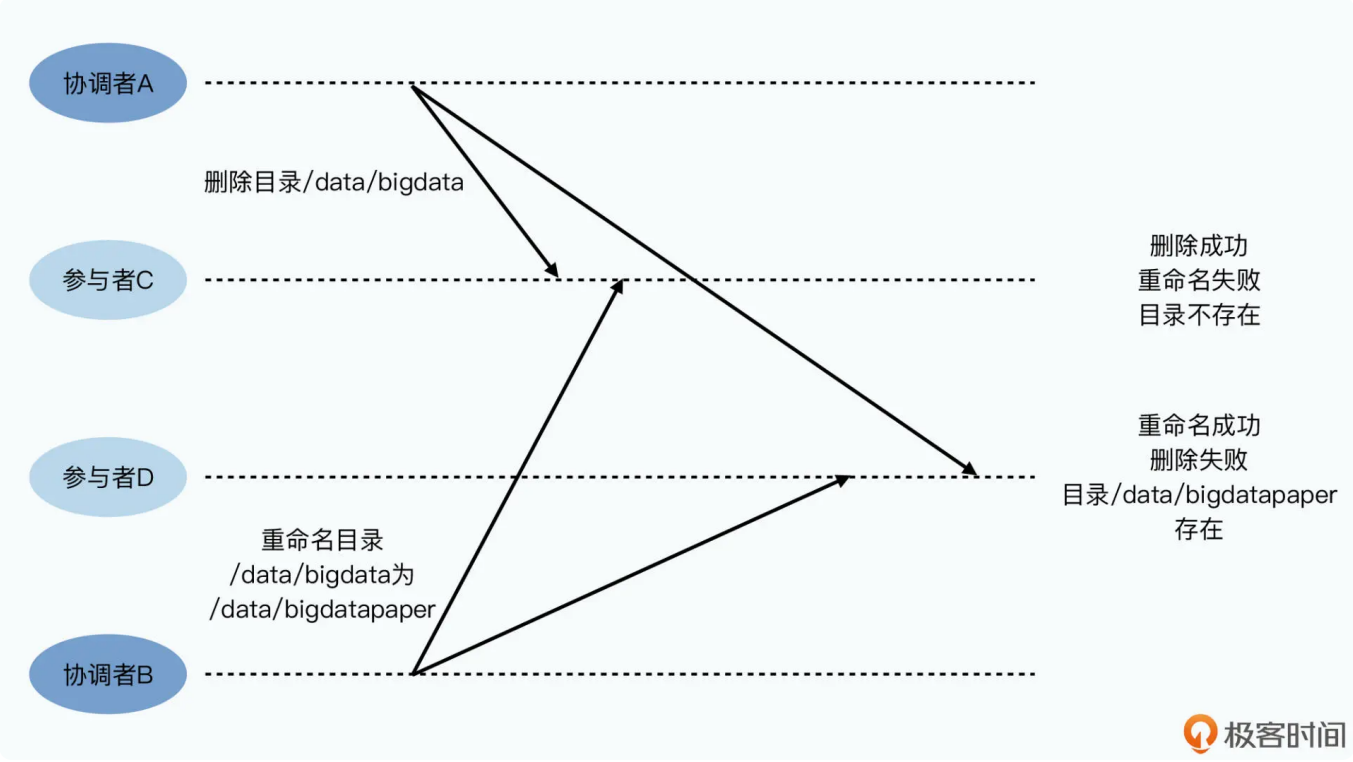
事实上，所有异步复制的主从系统，如果我们去读从库，都会遇到这样线性不一致的情况。

而为了保障线性一致性，或者说系统的可线性化，我们必须让主从节点之间是同步复制的。而要做到高可用的同步复制，我们就需要 **Paxos** 这样的共识算法。

并不可行的多协调者

我们先来回想一下上节课讲到的两阶段提交：我们可不可以，通过提供多个协调者来避免单点故障呢？比如说，当一个协调者挂掉的时候，我们可以切换到另外一个协调者，是不是就可以避免单点故障了，如果这样做会出现什么问题吗？

最容易发生的问题，是**操作顺序的错乱**。我们以两个协调者 A 和 B，以及两个参与者 C 和 D 为例，如果 A 要在 C 和 D 上，删除目录 /data/bigdata，而 B 则是要在 C 和 D 上，把目录 /data/bigdata 改名成 /data/bigdatapaper。因为是分布式的网络环境，那么可能会出现在 C 这里，A 的请求先到，B 的请求后到，/data/bigdata 的目录已经被删除了，所以改名也失败了；而在 D 这里，两个顺序是反过来的，那么 D 这里的 /data/bigdata 目录已经改名成功，而删除则失败了。C 和 D 之间的数据也就不一致了。



如果我们设置多个协调者，它们互相之间不沟通的话，就没法达成“共识”

所以，简单采用两个协调者的办法是行不通的。**核心挑战在于，多个协调者之间没有办法相互协调，达成一个两个操作在顺序上的共识。**而 Paxos，想要解决的就是这样的问题。

Paxos 算法中的协调过程

我们希望啊，我们在写入数据的时候，能够向一组服务器发起请求，而不是一个服务器。这组服务器里面的任何一个挂掉了，我们都能向里面的另外一台服务器发送请求，并且这组服务器里面的每一台，最终写入并执行日志的顺序是一样的。

在 Paxos 算法里，我们把每一个要写入的操作，称之为**提案（Proposal）**。接受外部请求，要尝试写入数据的服务器节点，称之为**提案者（Proposer）**，比如说，我们可以让一组服务器里面有 5 个提案者，可以接受外部的客户端请求。

在 Paxos 算法里，并不是提案者一旦接受到客户端的请求，就决定了接下来的操作和结果的，而是有一个**异步协调**的过程，在这个协调过程中，只有获得多数通过（accept）的请求才会被选择（chosen）。这也是为什么，我们通常会选择 3 个或者 5 个节点这样的奇数数字，因为如果是偶数的话，遇到 2:2 打平这样的事情，我们就没法做出判断了。

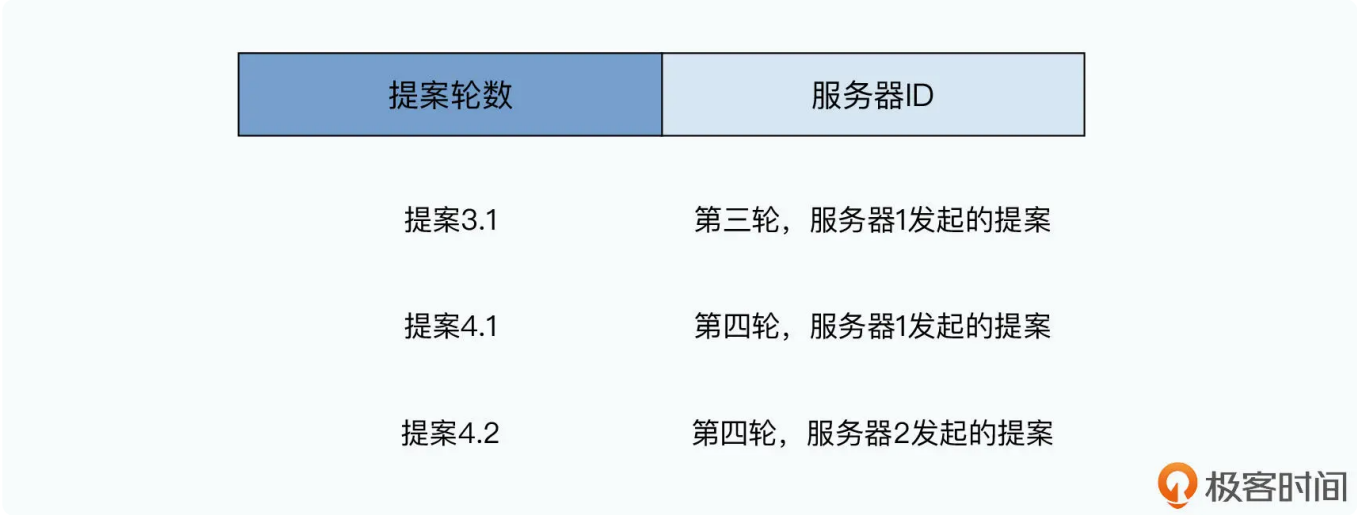
这个投票机制也是 Quorum 这个名字的由来，因为 Quorum 在英文里的意思就是法定人数。一旦达到了过半数，那么对应的请求就被通过了。

既然我们的提案者已经准备好 5 个节点了，我们不妨就复用这 5 个节点，让这 5 个节点也作为 Quorum，来承担一个叫做**接受者（Acceptor）**的角色。

给提案编号

首先是每一个请求，我们都称之为一个“提案”。然后每个提案都有一个编号，这个编号由两部分组成。**高位**是整个提案过程中的轮数（Round），**低位**是我们刚才的服务器编号。每个服务器呢，都会记录到自己至今为止看到过，或者用到过的最大的轮数。

那么，当某一台服务器，想要发起一个新提案的时候，就要用它拿到的最大轮数加上 1，作为新提案的轮数，并且把自己的服务器编号拼接上去，作为提案号发放出去。并且这个提案号必须要存储在磁盘上，避免节点在挂掉之后，不知道最新的提案号是多少。



提案号，是提案轮数和服务器ID的组合

通过这个方式，我们就让这个提案号做到了两点：

首先是不会有重复的提案号，不会存在两个服务器发出相同提案号的情况；

其次是提案号能够按照数值大小，区分出先后和大小。即使是同一状态下不同服务器发出的提案，也能比较大小。

Prepare 阶段

那么，当提案者收到一条来自客户端的请求之后，它就会以提案者的身份发起提案。提案包括了前面的提案号，我们把这个提案号就叫做 M。这个提案会广播给所有的接受者，这个广播请求被称为 **Prepare 请求**。

而所有的 Acceptor 在收到提案的时候，会返回一个响应给提案者。这个响应包含的信息是这样的：

首先，所有的接受者一旦收到前面的 Prepare 请求之后，都会承诺它接下来，永远不会接受提案号比当前提案号 M 小的请求；

其次，如果接受者之前已经接受过其他提案的内容（假设是 X）了，那么它要存储下已经接受过的内容和对应的提案号。并且在此之后，把这个提案号和已经接受过的内容 X，一起返回给提案者。而如果没有接受过，就把内容填为 NULL。

这样一个来回，就称之为 Paxos 算法里的 **Prepare 阶段**。要注意，这里的接受者只是返回告知提案者信息，它还没有真正接受请求。这个过程，本质上是提案者去查询所有的接受者，是否已经接受了别的提案。

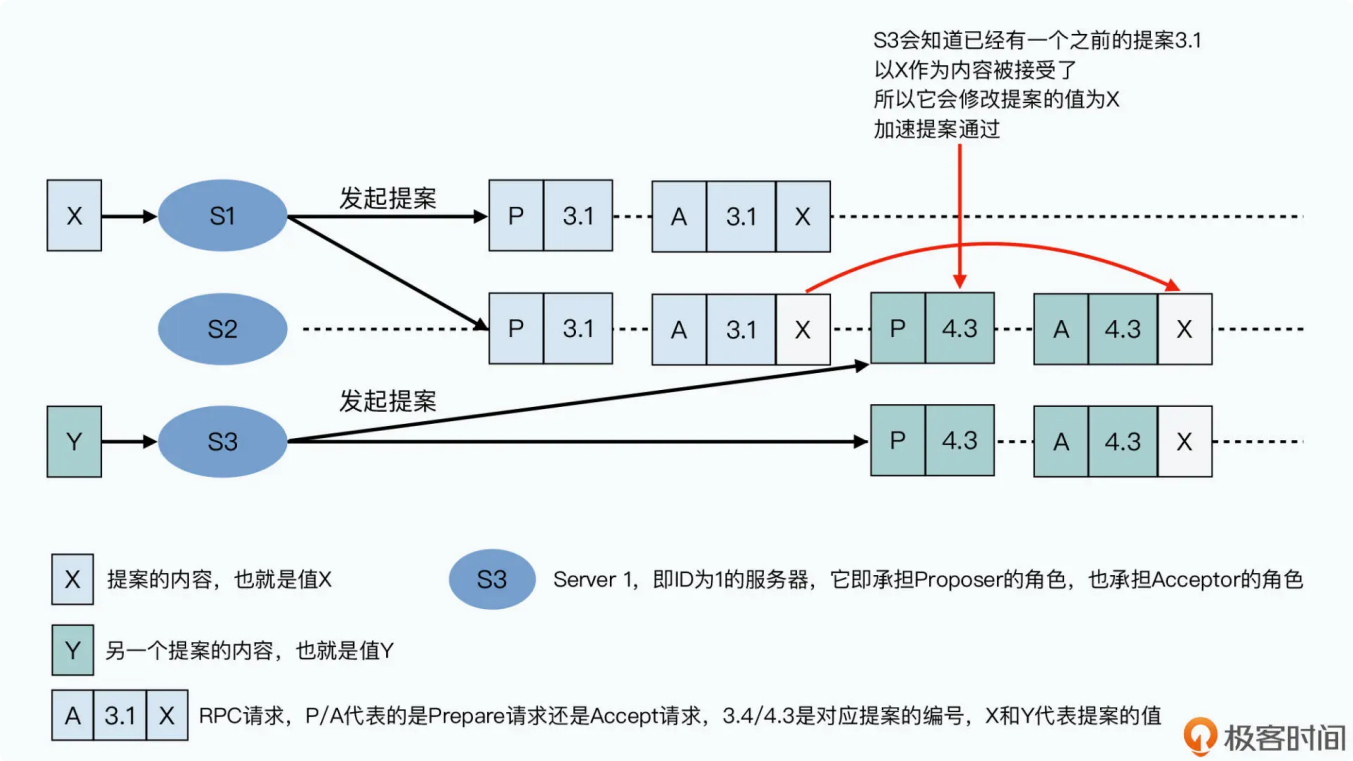
Accept 阶段

当提案者收到超过半数的响应之后呢，整个提案就进入第二个阶段，也称之为 **Accept 阶段**。提案者会再次发起一个广播请求，里面包含这样的信息：

首先仍然是一个提案号，这个提案号就是刚才的 Prepare 请求里的提案号 M；

其次，是提案号里面的内容，一般我们也称之为提案的值。不过这个值，就有两种情况了。

第一种情况，是之前接受者已经接受过值了。那么这里的值，是所有接受者返回过来，接受的值当中，提案号最大的那个提案的值。也就是说，提案者说，既然之前已经做出决策了，那么我们就遵循刚才的决策就好了。



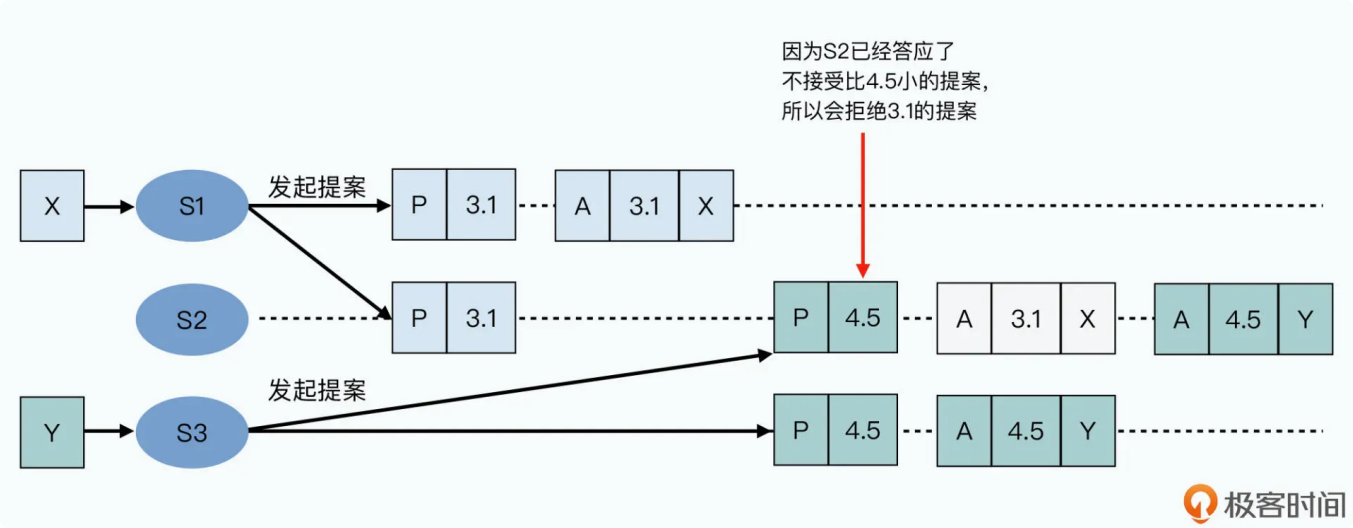
如果之前已经有接受者接受过值了，那么我们帮助之前的提案加速通过

而第二种情况，如果所有的提案者返回的都是 NULL，那么这个请求里，提案者就放上自己的值，然后告诉大家，请大家接受我这个值。

那么接受到这个 Accept 请求的接受者，在此时就可以选择接受还是拒绝这个提案的值。通常来说：

如果接受者没有遇到其他并发的提案，自然会接受这个值。一旦提案者收到超过半数的接受者“接受”的请求。那么它就会确定，自己提交的值被选定了。

但也有可能，接受者刚才已经答应了某个新的提案者说，不能接受一个比提案号 N 早的请求。而 N>M，所以这个时候接受者会拒绝 M。

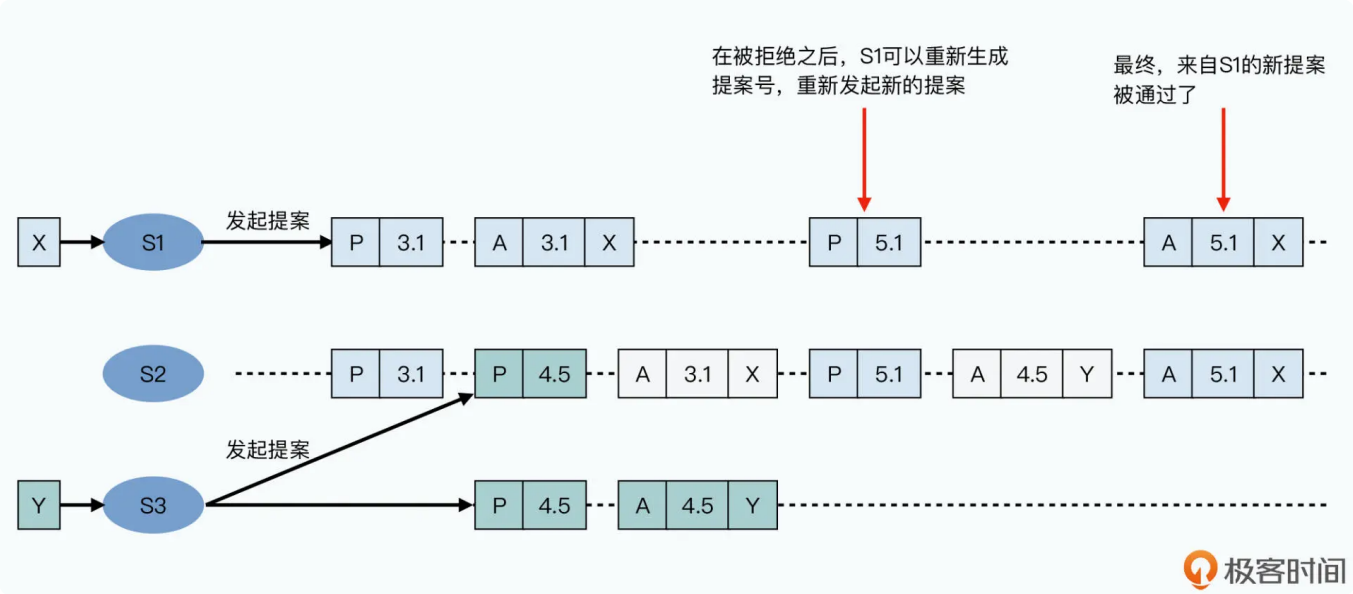


在Accept阶段，可能因为已经接受了更加新的Prepare的提案，会拒绝掉已经完成了Prepare阶段的提案

不管是接受还是拒绝，这个时候接受者都会把最新的提案编号 N，返回给提案者。

还是要注意，这个时候接受者接受了请求，并不代表这个请求在整个系统中被“选择”了。

提案者还是会等待至少一半的接受者返回的响应。如果其中有人拒绝，那么提案者就需要放弃这一轮的提案，重新再来：生成新的提案号、发起 Prepare 请求、发起 Accept 请求。而当超过一半人表示接受请求的时候，提案者就认为提案通过了。当然，这个时候我们的提案虽然没有变，但是提案号已经变了。而当没有人拒绝，并且超过一半人表示接受请求的时候，提案者就认为提案通过了。



如果有人拒绝，我们可以重新生成提案号，重新发起提案请求

可线性化和共识算法

可以看到，在 Paxos 算法这个过程中，其实一直在确保一件事情，就是**所有节点，需要对当前接受了哪一个提案达成多数共识。**

如果有多个 Proposer 同时想要向这个一致性模块写入一条日志，那么最终只会有一条会被成功写入，其余的提案都会被放弃。多个并发在多个 Proposer 上发生的写入请求，互相之间需要去竞争一次成功提案的机会。

通过 Paxos 算法，我们能够确保所有服务器上写入日志的顺序是一样的，但是我们并不关心是不是某个提案者会比另一个提案者先写入。毕竟这些请求都是在网络上并发进行的，我们也无法规定谁先谁后。

而在一次 Paxos 算法完成之后，被放弃的提案可以重新再来。这个节点如果还想要继续写入日志，可以在下一次 Paxos 算法的过程中，重新开始整个 Paxos 算法的过程。这样，在多个节点同时接受外部请求的环境下，我们不会出现多个节点之间，执行的日志顺序不一样的情况。

不过，相信你也发现了 Paxos 算法的一个问题，那就是**开销太大了**。无论是否系统里面出现并发的情况，任何一个共识的达成，都需要两轮 RPC 调用。而且，所有的数据写入，都需要在所有的接受者节点上都写入一遍。

所以，虽然 Paxos 算法帮助我们解决了单点故障，并且在没有单点的情况下，实现了共识算法，确保所有节点的日志顺序是相同的。但是，原始的 Paxos 算法的性能并不好。只是简单地写入一条日志，我们就可能要解决多个 Proposer 之间的竞争问题，有可能需要有好几轮的网络上的 RPC 调用。

当然，我们可以用各种手段在共识算法层面进行优化，比如一次性提交一组日志，而不是一条日志。这也是后续 Multi-Paxos 这些算法想到的解决方案。

但是，如果我们往一个数据库同步写入日志都要通过 Paxos 算法，那么无论我们怎么优化，性能都是跟不上的。**根本原因在于，在 Paxos 算法里，一个节点就需要承接所有的数据请求。**虽然在可用性上，我们没有单点的瓶颈了，但是在性能上，我们的瓶颈仍然是单个节点。

事实上，我们接下来要讲解的 Chubby，正是看到了 Paxos 算法的性能制约而搭建起来的。想要知道 Chubby 究竟是如何规避 Paxos 算法目前的性能瓶颈的，那就请你接着和我一起来学习下节课的内容吧。

小结

这节课，我们先一起来看了一下数据库事务的 ACID 特性，也就是原子性、一致性、隔离性和持久性。而我们真正关注的，是里面的**隔离性**。

为了让应用开发人员可以不用去理解复杂的并发，或者是自己加锁来解决业务问题，我们应该要让数据库支持“可串行化”的隔离级别，也就是所有的事务操作看起来，都是串行一个一个执行的。

不过，如果回顾 GFS 的 Master 和 Backup Master 的同步需求，会发现事务隔离性并不是我们的核心需求。

我们的核心需求是，Master 和 Backup Master 之间的数据是同步复制的。同步复制的高可用性方案，才能做到可线性化，而如果我们只有异步的主从复制，那么往往会在很多时候，在从服务器上读到过时的数据，无法反应我们刚刚进行的写入操作。

我们把 GFS 的 Master 和 Backup Master 之间的同步复制问题，当成了一个状态机日志复制问题，而解决这个问题的方案，就是 Paxos 这样的分布式共识算法。

我们通过这个算法实现了一个一致性模块，这个模块可以有多个节点接受外部请求。并且能够在多个节点并发“提案”的情况下，确保模块内各个节点的日志写入顺序是相同的，实现了状态机日志的同步复制功能。

通过这节课，我们理解了什么叫做事务里的可串行化，也明白了什么是分布式系统里的可线性化。在理解了 Paxos 这个分布式共识算法的实现过程之后，我们终于有了充分的基础知识了，那么下一讲，就让我们一起来深入了解一下，Chubby 这个系统是如何利用 Paxos 算法，来为 GFS 以及 Bigtable 实现高可用性的。

推荐阅读

Paxos 算法，是分布式共识中最主要的主题之一，我建议你一定要抽时间去读一读 Paxos 的相关论文，也就是《Paxos Made Simple》。而对于 Paxos 的算法讲解，比较深入详细的，是斯坦福大学约翰·奥斯特霍德（John Ousterhout）教授的这个讲座视频。

此外，《数据密集型应用系统设计》的第 7、8、9 三章，也对这个问题做了很多深入的探讨。我在上一讲里已经推荐过你去读一读，如果你还没有读过的话，那就现在赶紧去读一读吧。

思考题

按照惯例，最后还是给你留一道思考题。

我们所说的分布式事务的一致性，和分布式共识有相关性吗？那么可串行化和可线性化呢？为什么我们给它们取了非常接近的名字？这是一个很多文章、教程都没有给出清晰描述和解答的问题。在学完了这节课之后，你能谈谈你对这两个名词的理解吗？

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 分布式锁Chubby（一）：交易之前先签合同

下一篇 14 | 分布式锁Chubby（三）：移形换影保障高可用

精选留言 (8)

写留言

吴小智
2021-10-26

个人理解，在分布式的语境下谈一致性，指的是数据一致性和事务一致性两个方面；可串行化说的是数据一致性，写后要能读到；可线性化说的是事务一致性，数据库系统要保证事务执行后的结果与一个个执行事务是一致的。

展开

在路上
2021-10-25

徐老师好，分布式事务的一致性（Consistency），要从数据库事务ACID的特性之一，一致性去理解，指的是不同数据不会产生矛盾。分布式共识（Consensus），分布式系统的各个部分就按何种顺序处理数据达成一致意见的过程。也就是说一致性是结果，共识是达成一致的过程。可串行化（Serializable）是数据库事务中的概念，指的是事务被顺序处理，可线性化（Linearizability）是分布式中的概念，读操作可以读到最新写入的数据。以两…

展开

张建闯

2021-10-25

从数据库事务到状态机复制这个地方绕了好大一圈，本来没有这个疑惑，但是讲完这个完全给讲蒙圈了，看了两三遍才看明白，这两个是讲两个事情，为什么非要讲一下数据库事务呢？？

是我还是没有明白这两个的关联，还是，这个地方本来就没有什么关系呢

展开

webmin

2021-10-23

可串行化是指事物的隔离性，每个事物是串行执行，可串行化事物并不意味着在分布式多副本情况下每个事物按发生时间在副本上执行，只保证每次执行一个。
可线性化是说操作所有副本的行为与操作只有一个副本是一样的，且每个操作都是原子的，即对于任何两个操作，总是可以指出哪个操作在先。

展开

Geek_789fcf

2021-10-23

针对图有几处不明白：
1、既然S1，S2，S3代表服务器，那图里的备注“S3，server 1，即ID为1的服务器”，是不是应该写成"S3，server 3，即ID为3的服务器”
2、既然S1，S2，S3都是提案者，又是接受者，那为什么S1发起的提案，图中S3没有画出来...

展开

Geek_88604f

2021-10-21

在分布式系统中，分布式共识是分布式事务一致性的基础。还是以转账为例，首先通过分布式共识算法将谁给谁转账这个事实记录为日志，然后在遇到故障时通过这个日志进行重做或撤销，从而实现一致性。

2021-10-21

一个事务里会多次读取我们的银行卡余额。如果在这个过程中，我们在其他地方刷卡消费

了，那么两次读到的余额会不同。这个隔离级别一般也不太会使用，因为这样一个事务中两个完全相同的读操作，会读到完全不同的数据，也就是会出现“幻读”。------这个问题不是幻读，是不可重复读。RR隔离级别是解决不可重复读的，mysql innodb存储引擎也能同时解决有些情况下的幻读。可序列化才能解决幻读。

展开

Elon

2021-10-20

可线性化（Linearizability）和可串行化（Serializability）在英文环境中看着还挺像的，通过中文翻译了一次之后反而觉得还好。

串行化指在竞态条件（至少一个写操作）出现时，一次一个的执行操作。

...

展开

1